USTHB, Faculté Electronique et Informatique 25/5/2014
Département Informatique

Examen de POO

Exercice n°1 :

Un vecteur d'un espace vectoriel de dimension n est défini par n-uplet représentant les coordonnées du vecteur. Pour cela, on définit une classe **Vecteur** permettant de manipuler des vecteurs de dimension quelconque et caractérisée par un seul attribut qui est un tableau de coordonnées de type double nommé coord. (**private double[] coord ;**) Exemple v1. coord = { 2,7) ; v2.coord = { 2,0,7} et v3.coord ={0,0}.

1. Rajouter à cette classe les méthodes suivantes :
   a. un constructeur avec deux paramètres un entier n et un double x, qui crée le tableau de coordonnées à n éléments et les initialise à la valeur x.
   b. un constructeur avec un paramètre T de type tableau de double, qui crée le tableau de coordonnées à partir du tableau T.
   c. Une méthode **int taille()** qui retourne le nombre de coordonnées de l'objet Vecteur.
   d. Redéfinir la méthode **String toString()** afin qu'elle retourne les coordonnées du vecteur.
   e. Une méthode **void Afficher()** qui permet d'afficher les coordonnées du vecteur.
   f. Une méthode double **somCor()** qui retourne la somme des coordonné du vecteur courant.
   g. une méthode double **prodScal(Vecteur v)** qui retourne le résultat du produit scalaire du vecteur courant avec le vecteur v.
      On génère une exception si les deux vecteurs ne sont pas de même taille, (l'exception n'est pas traitée à ce niveau).
      On rappelle que le produit scalaire de deux vecteurs v1(x1,x2,x3,x4,…xn) et v2 (z1,z2,z3,z4,…zn) est égal à x1*z1+ x2*z2+ x3*z3+x4*z4+… + xn*zn.
   h. Redéfinir la méthode **boolean equals(Object o)** de telle sorte qu'elle retourne vrai ssi les deux vecteurs ont les mêmes coordonnées.
   i. Compléter la classe Vecteur pour définir un ordre entre les vecteurs en fonction de leurs tailles (nombres de coordonnées).
2. Rajouter à la classe Application les méthodes suivantes :
   j. Une méthode **void afficherList(ArrayList<Vecteur> list)** qui reçoit une liste dynamique de vecteurs et réalise ce qui suit :
      - Tri la liste par ordre croissant par rapport au nombre de coordonnées
      - Affiche tous les vecteurs de la liste qui sont égaux au vecteur V ayant les coordonnées (1,1,1).
   k. **int[] tabProd(Vecteur[] V1, Vecteur [] V2)** V1 et V2 sont de même dimension. Cette méthode retourne un tableau de produit scalaire entre les éléments de mêmes indices de V1 et V2.
      L'exception de la méthode proScal doit être levée. **(BONUS)**
3. Définir une classe **ArbreVecteur**, permettant de représenter un arbre binaire de recherche (arbre trié), l'arbre est caractérisé par un objet Vecteur et au plus deux fils (gauche et droit) qui sont eux même des arbres. Le tri de l'arbre est réalisé par rapport à la taille des vecteurs (nombre de coordonnées).
   - Rajouter à cette classe la méthode **ArrayList<Vecteur> listEgal (int s)** qui permet de retourner la liste de tous les vecteurs dont la somme des coordonnées est égale à s.

Nom : **Prénom :**

Exercice 2 (Répondre sur la feuille)

On désire simuler l'exécution des instructions machines, pour cela on définit les quatre classes suivantes :

```
1/ public class Processeur {
private  static HashMap<Integer, Integer>  memoire  = new HashMap();
private  static int compteurOrdinal;

public static Integer  getValeur (int adr)
{




  }

public static void  setValeur (int adr, int val)
{




  }

}
```

La structure memoire de type HashMap est composée de deux champs, le premier est la clé qui représente l'adresse de la variable et le deuxième la valeur de la variable. Les méthodes getValeur et setValeur permettent respectivement de récupérer et modifier la valeur de la variable.

```
2/ public abstract class Instruction {
private String codeOperation ;
private int adr1;
public Instruction(String code, int adr){ codeOperation = code ; adr1 = adr ;}




}
3/ public Incrementer extends Instruction {
public Incrementer(int adr) {super("INC", adr); }
```

Corrigé de l'examen de POO ISIL A + ISILB (2014)

Exercice 1 (13 points + 1)

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
public class Vecteur  implements Comparable{
private double [] coord;
// 0.5 points
public Vecteur(int n, double x){
        coord = new double[n];
        for(int i = 0; i<n; i++)
                coord[i]=x;
}
// 0.5 points
public Vecteur( double[]T ){
        coord = T;
}
// 0.5 points
public int taille() {
        return coord.length ;
}
// 0.5 points
@Override

public String toString(){
        String S= "";
        for( double v: coord)    S+= v+",";
        return S;
}
// On aurait pu le faire plus simplement
public String toString() {
        return "Vecteur [coord=" + Arrays.toString(coord) + "]";
}

// 0.5 points
public void afficher(){
        System.out.println(this);
}
// 1 point
public double somCor(){
        double S =0;
        for(double v: coord)     S+= v;
        return S;
}
// 1.5 points ( O.5  throw new Exception, O.5 throws, O.5 le reste
public double prodScal( Vecteur V) throws Exception{
        int PS=0;
        if(( V == null)|| ( taille() != V.taille() )) throw new Exception();
        for(int i =0;i<coord.length; i++) PS+= coord[i]*V.coord[i];
        return PS;
}
```

```
// 1 point
@Override
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (obj == null)
                return false;
        if (getClass() != obj.getClass())
                return false;
        Vecteur V = (Vecteur) obj;

        for(int i = 0; i<coord.length; i++)
        if( coord [i] != V.coord[i]) return false;
        return true;
        // on aurait pu faire simplement if (!Arrays.equals(coord, V.coord))

}
// 1 point
@Override
public int compareTo(Object obj) {
        Vecteur V = ( Vecteur) obj;
        return taille() - V.taille();
}

}
class Application{
        // 3 points ( 0.5 + 0.5 +1.5+ 0.5
        public void afficherList( ArrayList< Vecteur> list){
                Collections.sort(list);
                Vecteur V1 = new Vecteur(3,1);
                for( Vecteur V2: list){
                        if(V2.taille()>3)break;
        if(V2.equals(V1))V2.afficher();
                }
        }
  // 1 point Bonus
  public double[] tabProd ( Vecteur[] V1,Vecteur[] V2 ){
          double[] T = new double[ V1.length];
          for( int i = 0; i< V1.length; i++)
                try {       T[i]= V1[i].prodScal(V2[i]);}
          catch ( Exception ex){ System.out.println("Produit scalaire impossible : tailles différentes");}

          return T;

          }
  }
class ArbreVecteur {
        private Vecteur racine;
        private ArbreVecteur fg,fd;
        public ArbreVecteur(Vecteur racine) {
                this.racine = racine;
        }
        // 3 points
        public ArrayList<Vecteur> listEgal(int s){
                ArrayList<Vecteur> list = new ArrayList();
```

```
            if(racine.somCar()==s)list.add(racine);
            if( fg != null) list.addAll(fg.listEgal(s));
            if( fd != null) list.addAll(fd.listEgal(s));
        return list ;
        }}
```

EXERCICE 2 (7 points)

```
import java.util.HashMap;
// 2 points
public class Processeur {

private static HashMap<Integer,Integer> memoire = new HashMap();
private static int compteurOrdinal;

public static Integer getValeur( int adr) throws Exception{
        Integer val = memoire .get(adr);
        if( val == null) throw new Exception("Adresse Non existante");
        return val;
}

public static void setValeur( int adr, int val) throws Exception{

        Integer  v= memoire .get(adr);
        if( v == null) throw new Exception("Adresse Non existante");
memoire.put(adr,val);
}

public static void setCompteurOrdinal(int adr){
        compteurOrdinal = adr;
}
}
// 1 point
public abstract class Instruction {
private String codeOpération;
private int adr1;
public Instruction(String codeOpération, int adr1) {
        super();
        this.codeOpération = codeOpération;
        this.adr1 = adr1;
}
public int getAdr1() {
        return adr1;
}


public abstract void evaluer() throws Exception ;

}
// 1 point
public class Incrementer extends Instruction {

        public Incrementer(int adr) {
                super("INC", adr);


}
```

```
        @Override
        public void evaluer() throws Exception {
                int val = Processeur.getValeur(getAdr1());
                val++;
                Processeur.setValeur(getAdr1(), val);
}}

// 1 point
class SeBrancher extends Instruction {

        public SeBrancher(int adr) {
                super("JUMP", adr);

}

        @Override
        public void evaluer()  {
                Processeur.setCompteurOrdinal(getAdr1());
        }
}
//1 point
public class Additionner extends Instruction {
      private int adr2;
        public Additionner(int adr1, int adr2) {
                        super("ADD", adr1);this.adr2= adr2;
                        }

                @Override
                public void evaluer() throws Exception {
                        int val = Processeur.getValeur(getAdr1());
                        val +=  Processeur.getValeur(adr2);

                        Processeur.setValeur(getAdr1(), val);


                }
}

// 1 point
import java.util.ArrayList;
public class Application {

void evaluerProgramme(ArrayList<Instruction> list) throws Exception{
        for( Instruction inst : list )
                inst.evaluer();'
}
}
// Le concept utilisé est le polymorphisme il permet de déterminer dynamiquement la méthode ( code) la plus
spécifique à invoquer selon l'instanciation de l'objet. Son intérêt apparait  plus particulièrement dans le cas où
type de la déclaration d'un objet (liaison  statique) est différent de type de son instanciation (liaison dynamiq
```